Coordination and agreement

- Coordination:
  - Given a set of processes, we want them to agree on some value.

- Instances of coordination problems occur all over the place in distributed systems.
  - What is the balance of the bank account?
  - Who has control over modifying the database?
  - Which version of the data is up to date?

Failures

- Before proceeding, we must explicitly state what failure model we will assume.

- In discussing coordination algorithms, we assume:
  - A reliable communication channel.
    - In some algorithms, we also assume that this channel supports group communications.
  - Processes may fail.
  - We may attempt to decide if a process has failed.

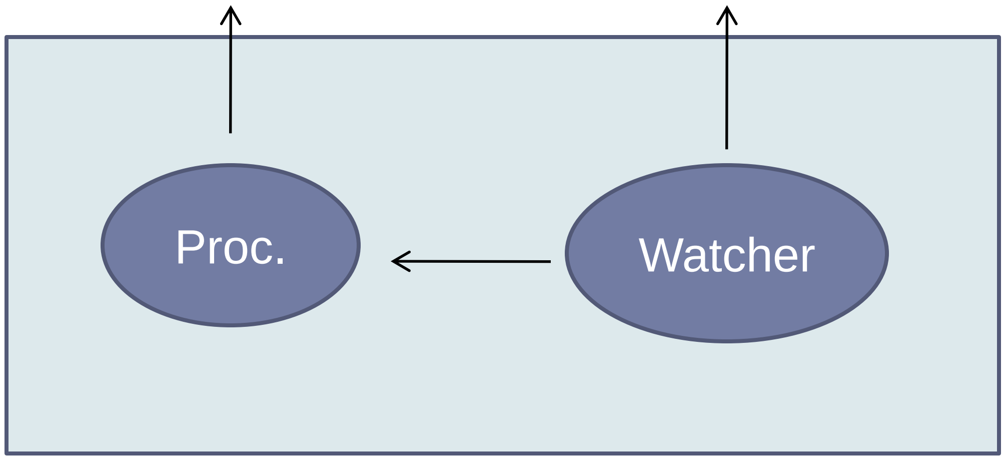
Failure detection

- If we want to say whether or not a process has failed, we need a detection mechanism that determines the state that the process in question is in.

- Unreliable mechanism
  - *Unsuspected*: “We recently heard from it.”
  - *Suspected*: “Process may have failed – we haven’t heard from it in a while.” (Timeout)
- Reliable mechanism
  - *Unsuspected*: “We recently heard from it.”
  - *Failed*: It is dead.

Failure detection

- Obviously failure detection is a hard problem that is somewhat circular.
- One mechanism is to place a watcher in a node that has the sole purpose of monitoring the processes.
  - If the watcher watching a process dies, but the process itself lives, how do we conclude that it has failed or not?



A fundamental coordination problem

- In any concurrent system, be it distributed, a shared memory parallel system, or even a concurrent multitasking platform, this problem comes up.

- How do we control access to code and/or data that, for correctness reasons, only one process can work with at any given time?

- This is called the *mutual exclusion* problem.

Mutual exclusion

- What is an example of this? Say we have a bank account with primitive operators like “withdraw” and “deposit”.
- Now, say we write a transfer operation like this:

```
Transfer(amt, src, dest) =
  src_balance = src.getBalance();
  src_balance -= amt;
  dest_balance = dest.getBalance();
  dest_balance += amt;
  dest.setBalance(dest_balance);
  src.setBalance(src_balance);
```

- What could possibly go wrong?

Mutual exclusion

- Easy. We could have two processes read the balance before one of them has written the modified balance back to the store.
- We want to ensure that only one process is allowed to be in the transfer operation at any given time.
- We'd call the transfer code to be a *critical section*.

- How do we do this in a distributed system?
  - In a sequential system, we typically rely upon locks and atomic operations to protect critical sections.
  - We don't have those in a distributed system where we assume that all we have are messages.

Distributed mutual exclusion

- Consider the following to be the abstract structure of a critical section that we want to enforce:

```
Enter()
Do_Dangerous_Stuff()
Exit()
```

- So, what we need to implement is a distributed mechanism with which a process can determine if it is safe for it to enter the region, and notify others when it is out.

Requirements

- **Safety**: At most one process may be in the critical section at a given time.
  - Safety = Critical section protected.
- **Liveness**: Requests to enter or exit eventually succeed.
  - Liveness = Nobody sits forever waiting to go in.
- **Fairness**: Ordering of entries into enter/exit request queues respected when granting access.
  - Fairness = Everyone gets a fair chance, nobody can cut in line.

Evaluation criteria

- Given an algorithm to implement mutual exclusion in a distributed system, we can measure:
  - How many messages must be sent at entry and exit.
  - What delay is incurred by a process at entry and exit.
  - Throughput: The rate at which the set of processes can process the critical section.

Important Assumption

- An underlying assumption here is that the processes who wish to enter the critical section obey the entry and exit protocol.
- If a client sees that the critical section is occupied, and goes ahead anyways into it, there is little these protocols can do to prevent it.
- We assume that the processes are correct – in other words, they obey the protocol.

Algorithm 1: Central server

- This is the easiest method.

- Central server controls an access token.
  - When a process wants to enter critical section, it asks the server for the token.
  - If the server has it, it gives it to the process requesting it.
  - The process returns the token when it exits the critical section.
  - Any process that arrives when the token is not on the server wait in a FIFO queue.

Algorithm 1: Central server

- Two messages to enter the critical section, so delay due to messaging.
  - Even if no other process is in the critical section.
- One message to exit it.
  - Asynchronous message would eliminate delay on exit.

- Critical point here is the server. Many clients can cause it to slow down, and if the rate of requests to the CS is faster than any client executing it, we can see the FIFO of clients back up.

Algorithm 2: Token ring

- A token circulates amongst a set of processes.
- When a token arrives:
  - If the receiver is waiting to enter the CS, enter it. Pass the token along when the exit occurs.
  - If the receiver is not waiting to enter the CS, just pass the token along.
- So, if a process wants to enter the CS, just wait for the token to arrive.

Algorithm 2: Token ring

- Continuous network resources required.
  - Say no process wants to enter the CS for a long time. In the meantime, the token circulates.
- Worst case delay to get into the CS? A process just passed the token along right before it wanted to enter the CS. Has to wait for N message transfers to occur.
  - Delay ranges from 0 (token is arriving) to N message transfers.
- One message on exit, when the token passed along.

Algorithm 3: Ricart & Agrawala

- Multicast based algorithm

```
On initialization
    state := RELEASED;
To enter the section
    state := WANTED;                            }
    Multicast request to all processes;         }  ←  request processing deferred here
    T := request's timestamp;                   }
    Wait until (number of replies received = (N – 1));
    state := HELD;

On receipt of a request <$T_i$, $p_i$> at $p_j$ ($i \neq j$)
    if (state = HELD or (state = WANTED and ($T$, $p_j$) < ($T_i$, $p_i$)))
    then
        queue request from $p_i$ without replying;
    else
        reply immediately to $p_i$;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;
```

Algorithm 3: Ricart & Agrawala

- Performance
  - N messages : One to request entry, N-1 to receive replies.
  - May be higher if multicast isn’t supported in hardware.
- Delay is related to round trip time.

Algorithm 4: Maekawa’s Voting Alg.

- Based on the idea that clever partitioning of the processes into “voting sets” allows granting of access to critical section from only a subset of processes.
  - The trick is that every subset has a non-empty intersection with every other.
- We’ll walk through this over two slides – it’s a bit more complex, but quite clever.

Algorithm 4: Maekawa’s Voting Alg.

- Initialization:
  - Everyone sets their state to released.
  - Everyone sets their voted flag to false.

- For a process p to enter the critical section:
  - Sets it’s state to wanted.
  - Multicast request to all processes in it’s voting set.
  - Wait until it hears back from everyone in it’s voting set.
  - Set it’s state to held when this occurs, and proceed into the CS.

Algorithm 4: Maekawa’s Voting Alg.

- On receipt of message from p on q.
  - If q is in a held state or has voted
    - Queue the request from p, don’t reply.
  - Otherwise
    - Reply to P
    - Set voted flag to true.

- When p exists CS:
  - Set state to released
  - Multicast release to all members of voting set

- On receipt of release from p on q.
  - If queued requests exist
    - Pop the queue, send a reply to it.
    - Set voted to true.
  - Otherwise
    - Set voted to false.

Algorithm 4: Maekawa’s Voting Alg.

- The benefit of this algorithm is that Maekawa showed the optimal size of the voting sets is related to sqrt(N).

- So the number of messages is reduced to O(sqrt(N)), which is preferable to the other O(N) or worse solutions.

- Delay again related to round trip time.

Algorithm 4: Maekawa's Voting Alg.

- This algorithm has a potential problem.
  - Deadlock!
- V[1] = {p1, p2}. V[2] = {p2,p3}. V[3] = {p3,p1}
- Three processes concurrently ask for entry into the CS.
- Algorithm can be fixed though.

Elections

- **Election**: Picking some process to play a special role (“server” or “master”) from a set of peers.

- We saw an instance of this previously in the Berkeley time algorithm.
  - One machine is tagged as the server, and it coordinates synchronizing a set of peers.
  - If that machine goes away, the set of peers can elect a new machine to play server.

Elections

- Elected process is unique.
  - If two processes call an election at the same time, only one process will end up being the winner.

- The result of an election is the process with the largest identifier.
  - Identifier is some arbitrary, but likely useful piece of information, like 1/load.

Election requirements

- Safety
  - Each process will either have an undefined elected peer, or all will have the same non-crashed process at the end of the election with the largest identifier.
- Liveness
  - All processes will eventually determine who was elected, or crash.

Algorithm 1: Ring election

- All processes start as non-participants.
- One process calls the election.
  - Marks itself as a participant.
  - Sends an election message clockwise with it’s identifier attached.
- Receivers compare the content of the message with their own identifier.
- If it is greater
  - Forward the message
- Otherwise
  - If not already a participant
    - Substitute local identifier and pass on
  - Otherwise
    - Don’t forward the message.
  - Either way, mark self as a participant.

Why?

Algorithm 1: Ring election

- If the identifier of the received message is the same as the receiver, then receiver knows it was greatest.
  - Mark self as non-participant.
  - Send elected message with it’s identity.

- Receive elected message
  - Set self as non-participant.
  - Store elected identifier.
  - Forward it if it wasn’t the originator of the elected message.

Algorithm 2: Bully algorithm

- The bully algorithm allows for crashes.
  - The ring algorithm doesn’t tolerate these very well since everyone only knew their neighbors.
- Synchronous model assumed, with timeouts to detect failures.
- The basic idea is that instead of the neighbor only model of the ring, every process knows the processes above it assuming everyone has a unique identifier with a well defined order (<).

Algorithm 2: Bully algorithm

- A process who wants to start an election sends an election message to all processes above it.
  - If it receives an answer from at least one, it is not the coordinator.
- A process that receives the election message answers and attempts to contact those above it.
- This proceeds until a process receives no answers.
  - Either due to timeouts and process failures, or it being the maximum.
- This allows a single process to decide that it is coordinator.
  - Of course, we have to deal with slow processes. What happens if a process doesn't fail but just goes slow in answering?